関数型プログラミング言語「F#」による統計計算の実際

福井昭吾

概　要

関数型プログラミング言語において、関数は値として扱われる。これにより、柔軟な関数の定義や、関数の引数として関数を与えることが可能となる。本稿で取り上げた統計計算では、関数型プログラミングの仕組みを使用することで、従来のプログラミング言語と比較して効率的な記述が可能となった。関数型プログラミング言語の利用により、プログラミング速度の向上、ソースコードの可読性および保守性が向上する可能性がある。

abstract

In the functional programming languages, function is a value with functional type. Thereby a function is defined flexibly and passed to the other functions easily. For statistical computing, we can write codes efficiently with functional programming compared to commonly used programming paradigm. We could improve the speed of coding, readability and maintainability of the source codes by the functional programming languages.

1　関数型プログラミング言語とは

関数型プログラミング言語とは、関数がファーストクラスであるプログラミング言語である。MSDNによれば、「関数がファーストクラスである」とは、(1)関数に名前をつけることができる、(2)関数を配列やリストなどのデータ構造に格納できる、(3)関数を引数として渡すことができる、(4)関数を戻り値として返すことができる、という基準を満たすことを意味する[*1]。つまり、関数型プログラミング言語では、関数を値として扱うことができるのである。現在利用されているプログラミング言語の多くは、数値、

＊本稿において記載される製品名および会社名は、各社の商標または登録商標である。

＊1　MSDNは、Microsoft社の開発者向けウェブサイトである（http://msdn.microsoft.com/ja-jp/ms348103）。

配列やリスト、および、クラスの実体としてのオブジェクトをファーストクラスとしているが、関数をファーストクラスとするものは少ない。したがって、関数がファーストクラスであるという点が、関数型プログラミング言語の大きな特徴であるといえる。

関数型プログラミング言語の歴史は古い。Hutton（2009）によれば、1930年代に関数型プログラミング言語の理論的基礎である「λ計算」が考案され、1950年代には最初の関数型言語とされる「Lisp」が作られたとされる*2。その後、(O)Caml, Haskell, Scala, F# といった関数型プログラミング言語が開発されている。

以前は、命令型言語やオブジェクト指向言語と比較して、関数型プログラミング言語が実際に使用される機会は少なかった。その理由として、関数型プログラミング言語は、手続き型言語等と比較してプログラムの記述が大きく異なっており、それが関数型プログラミング言語の習得を困難なものとしていたと考えられる。また、他の言語と比較して、関数型プログラミング言語の処理系が少なかったという点も挙げられる。

近年、関数型プログラミング言語は大きな注目を集めている。その理由の一つとして数値計算の効率化が挙げられる。科学技術計算だけでなく、実際のソフトウエア開発においても、複雑な数値計算がしばしば要求される。関数型プログラミング言語を導入することにより、これらの計算の効率的な記述が期待されている。ただし、ここで言う「効率化」は開発・記述効率の向上であって、演算処理速度の上昇ではない点に注意しなくてはならない。これまでのプログラミング言語におけるパラダイムの変化の多くがそうであったように、関数型プログラミング言語もまた、プログラミング効率の向上を目的としている。

プログラミング言語の中には、関数型プログラミングをサポートする（あるいは、新たにサポートし始めた）ものも多い。例えば、オブジェクト指向

---

*2　λ計算の理論的な詳細については、高橋（1991）を参照。

プログラミング言語の一つである「C#」は、バージョン2.0から関数型プログラミング言語の一部機能を取り込んでいる。また、統計計算ソフトウエア「R」はベクトル型言語を標榜しているが、実際には関数型プログラミングをサポートしている。数値計算ソフトウエアである「Mathematica」も関数型プログラミングによる記述が可能である。

本稿では、主に統計計算を例に挙げて、関数型プログラミング言語の有用性について述べていく。以下で実際に使用する言語は「F#」である*3。F#は、Microsoft社によって開発されたプログラミング言語で、関数型プログラミングだけでなくオブジェクト指向プログラミングをサポートした、いわゆるマルチパラダイム言語である。そのため、命令型プログラミング言語やオブジェクト指向プログラミング言語といった、現在広く使われている言語の機能を組み合わせつつ関数型プログラミングを行うことができる。したがって、純粋な関数型プログラミング言語と比較した場合、F#の学習にかかる手間は軽減されうるという利点がある。

比較のために用いる言語として、以下では「C#2.0」を用いる。ただし、C#2.0では関数型プログラミングを一部サポートしているため、その機能を使わずにプログラムの記述を行う*4。

2　F#の基礎（結合と型推論）

関数型プログラミング言語は、「関数がファーストクラスである」以外にも様々な特徴を持つ。これらの特徴について、実際の開発例を挙げながら説明していこう。

ソースコード1は、標本から度数分布を計算するF#のプログラム例である。

---

*3　F#の仕様は、"The F# 2.0 Language Specification" としてWeb上で公開されている（http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.html）。また、Syme et al.（2010）は、F#の機能について実例を用いて詳細に説明している。

*4　C#によるプログラミングの実際については、川俣（2009）を参照。

ソースコード1 度数分布の導出

```
 1 let xdata = [1.0; 2.0; 2.5; 3.0; 4.0; 5.0; 6.0; 10.0]
 2 let xbreaks = [1.0; 2.0; 3.0; 5.0]
 3
 4 let FreqDist data breaks lcomp rcomp =
 5     let(mins, maxs)=((-infinity)::breaks, breaks @
           [infinity])
 6     let classid = [1..(breaks.Length + 1)]
 7     let classinfo = List.zip3 mins maxs classid
 8
 9     let cidsearch x =
10         let(cmin, cmax, cid)=
11             List.find(fun(min, max, id) -> (lcomp min x)
                   && (rcomp x max))classinfo
12         cid
13     let classified = List.map cidsearch data
14
15     let freq cid = List.fold(fun acc id -> if id = cid
       then acc +1 else acc)0 classified
16     List.map freq classid
17
18 let res = FreqDist xdata xbreaks(<)(<=)
19 // val res : int list = [1; 1; 2; 2; 2]
```

FreqDist関数は、引数として標本（data)、階級の区切り（breaks)、各階級の下限を判定する関数（lcomp)、および各階級の上限を判定する関数（rcomp）を受け取り、度数分布を計算する。FreqDist 関数の処理の流れは以下の通りである。初めに、各階級の下限・上限・番号のセットを、リスト

にして一つの変数にまとめる（5-7行目）。階級の番号は、小さい階級から1, 2, . . . と割り当てている。次に、FreqDist関数内でcidsearch関数を定義する。cidsearch関数は、引数として標本内の数値一つを受け取り、それがどの階級に属するかを判定し、所属する階級の番号を戻り値として返す（9-12行目）。このcidsearch関数を使い、data内のすべての値について所属する階級の番号を取得し（13行目）、階級の番号ごとに、その個数を合計し度数分布を求めている（15, 16行目）。18行目では、実際にFreqDist関数を使っている。標本として1行目で定義している変数xdataを、階級の区切りとして2行目で定義している変数xbreaksを与えている。また、各階級の下限を判定する関数として演算子「<」を、各階級の上限を判定する関数として演算子「≤」を与えている。これにより、階級kの下限をx_k^l，上限をx_k^rとするとき、各階級は半開区間$(x_k^l,\ x_k^r]$となる。

一般的なプログラミング言語と関数型プログラミング言語との大きな違いとして、結合（binding）がある。関数型プログラミングで変数に値を与える場合、結合を用いる。結合は変数とデータとを結びつけるもので、F#ではletキーワードを使って行われる。例えば、上記プログラム例の6行目では、変数classidに対して整数列からなるリストを結合している。一方、9-12行目では、変数cidsearchに対して関数を結合している。

結合は、一般的なプログラミング言語での「代入」と等価ではない。一般的なプログラミング言語では、データの型が一致する限り、変数に対してデータを再度代入することができる。一方、関数型プログラミング言語における結合は、変数とデータと結びつけるものであり、結合された変数に対して別のデータを結合することはできない。例えば、F#では、ソースコード2を実行するとエラーが発生する。

ソースコード2　変数への再結合によるエラー

```
1 let x = 3.14
2 let x = 1.41 // error: value 'x' の定義が重複しています
```

結合は、副作用（side effects）によるエラーを防ぐことを目的としている。変数に代入されたデータが変更可能である場合、予期しない時点で変数の中身が変更されることでエラーが発生する可能性がある。F# では、変数と結合するデータの変更を禁止し、このようなエラーの発生を防いでいる。

結合により副作用によるエラーを防ぐことができる一方、従来型のループ処理の記述が難しくなる。for ループや while ループといったループ処理は、「ループの中で変数の中身が変更され、その変数が一定の条件を満たしたときにループが終了される」という流れである。F# で、再代入可能な変数を使わず結合のみを使うならば、変数の中身を変更することができないため、このようなループ処理は極めて困難なものとなる。したがって、上記の状況においてループ処理を行う場合、再帰やリスト関数が使われる。

ソースコード1ではループ処理の代わりに、リスト関数を使用している。リストとは、複数のデータを一つにまとめたものである。リスト内の各データは順序が指定されており、リストへの新しいデータの追加や、既存のデータを削除することは可能であるが、リスト内の各要素のデータを変更することはできない。

リスト関数は、リストに対して様々な処理を行う一連の関数である。例えば、リスト関数の一つである List.find メソッドは、「引数であるリストの各要素に対して、もう一つの引数で指定した判定関数を適用し、最初に判定を満たしたものを返す」というものである。これにより、for ループなどを用いることなく、リスト内のデータごとに条件判定を行うという反復処理が可能となる。上記のプログラム例では、他にも List.zip3, List.map, List.fold といったリスト関数が使用されている。いずれも、従来型の言語であればループ処理によって記述される部分である。

実際には、関数型プログラミングの機能を持つ言語は、マルチパラダイム言語であることが多く、F# でも、再代入可能な変数や、従来の言語にあるようなループ（for ループ、while ループ）が利用可能である。ただし、これらを用いる場合、当然ながら副作用に伴うエラーの発生について考慮しな

ければならない。

F# の別の大きな特徴として、型推論が挙げられる。一般的なプログラミング言語では、変数を使うときに型を指定する必要がある。例えば、C#2.0で変数 x に整数100を代入して使う場合には、「int x = 100」と記述する。この命令では「x は整数を代入できる変数である（x は整数型である）」と指定し、整数値100を代入している。一方、F# において、変数や引数の型は、計算の過程でそれらが利用されている状況から推論される。これを型推論と呼ぶ。例えば、FreqDist 関数内の classid 変数は型が指定されていない（6行目）。しかし、それに続く処理が「1から（breaks 内の要素数 + 1）までの整数をリストにして、代入する」ものであるため、「classid は整数型のリストである」と推論されるのである。

F# では、変数や引数に対して明示的に型を指定することも可能である。特に、型の推論が正しく行われない場合は明確に型を指定する必要がある。

3　F# における関数

別の例として、標本から、平均や分散などの代表値を計算するプログラムを作成することを考える。代表値の計算は、「標本の各数値に対して何らかの変換を行い、」「それらの合計を標本の大きさで割る」という一連の計算を含むことが多い。例えば、標本を $x_i (i = 1, \ldots, n)$ とすると、平均の計算は、

$$\bar{x} = \frac{\sum f(x_i)}{n}, f(x) = x,$$

分散の計算は、

$$s^2 = \frac{n}{n-1}\frac{\sum f(x_i)}{n}, f(x) = (x - \bar{x})^2$$

となる。

これらの計算をプログラミング言語で実装することを考えた場合、共通する部分である

$$\frac{\sum f(x_i)}{n} \tag{1}$$

をStatCommon関数として定義する、という方法が考えられる。StatCommon関数は引数として標本x_iと関数$f(x)$を受け取り、上の式から得られた計算結果を戻り値として返すものとする。平均を計算する関数（Mean関数）や分散を計算する関数（Variance関数）は、その内部でStatCommon関数を使う。例えばVariance関数であれば、StatCommon関数に標本と関数$f(x)=(x-\bar{x})^2$を渡し、その結果に$n/(n-1)$を掛けることになる。

プログラミングでは、複数の計算で共通する部分を一元化するよう設計することが望ましいとされる。これにより、プログラムの可読性が向上し、バグの発生を抑えることができる。さらに上の例では、別の代表値を計算する場合、追加的な記述が少なく済む。例えば、3次の平均周りの積率である歪度を計算したいならば、StatCommon関数に引数$f(x)=(x-\bar{x})^3$を渡すよう記述すれば良い。

ソースコード3はC#2.0によるプログラム例である。

ソースコード3　平均および分散の計算（C#2.0）

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.Text;
 4
 5 namespace CSharpExample
 6 {
 7     public static class Statistics
 8     {
 9         private static List <double> Data;
10         private delegate double Conversion(double x);
```

```
11
12          public static double Mean(IList <double> data)
13          {
14              return StatCommon(data, MeanConversion);
15          }
16
17          public static double Variance(IList <double> data)
18          {
19              Data = new List <double>();
20              Data.AddRange(data);
21              double n = data.Count;
22              return StatCommon(data, VarianceConversion)*
                    (n/(n - 1));
23          }
24
25          private static double StatCommon(IList <double>
                data,Conversion f)
26          {
27              double sum = 0.0;
28              for (int i = 0; i < data.Count; i++)
29                  sum += f(data[i]);
30
31              return sum / data.Count;
32          }
33
34          private static double MeanConversion(double x)
35          {
36              return x;
```

```
37          }
38
39          private static double VarianceConversion(double x)
40          {
41              return Math.Pow((x - Mean(Data)), 2.0);
42          }
43      }
44
45      class Program
46      {
47          static void Main(string[] args)

48          {
49              double[] data = new double[5] {1.0, 2.0, 3.0,
                    4.0, 100.0};
50
51              Console.WriteLine(Statistics.Mean(data));
52              // 出力：22
53              Console.WriteLine(Statistics.Variance(data));
54              // 出力：1902.5
55          }
56      }
57 }
```

以下、C# のプログラムにおいては、クラス内の変数を「メンバ変数」、クラス内の関数を「メソッド」と呼ぶ。

このプログラムは、統計計算に関する機能をまとめたStatisticsクラスと、Mainメソッドを含むProgramクラスからなる。このプログラムの開始地点（エントリポイント）は、Mainメソッドである。Mainメソッドでは、

Statistics クラスを使って平均と分散を計算している。一般的な数値計算ソフトと同じく、Mean メソッドと Variance メソッドの引数 data に標本を渡すことで平均と分散が計算される。(51・53行目)。Statistics クラスの Mean メソッドと Variance メソッドは、与えられた引数 data に基づいて計算を行う。また、Statistics クラスには、標本を一時的に保持するメンバ変数 Data が存在する。StatCommon メソッドは上述の通り式（1）に相当し、標本を受け取る引数（data）と関数$f(x_i)$を受け取る引数（f）を持つ。

このプログラムには、二つの特徴がある。

第一に、関数の引数として関数を渡すために、多くの記述が必要とされている。C# では、関数型プログラミングの機能を使わなくても、「デリゲート」を用いることで、関数の引数として関数を渡すことが可能である*5。デリゲートとは、C# における型の一つである。デリゲートのインスタンスには、そのデリゲートのシグネチャ（戻り値と引数の個数・型）と一致する関数を代入することができる。上述のプログラム例では、Statistics クラス内で Conversion デリゲートが定義されている（10行目）。Conversion デリゲートは、double 型の戻り値と double 型の引数１個からなる。StatCommon メソッドは、引数として Conversion デリゲートを受け取る。これは、式（1）の$f(x_i)$に相当している。Mean メソッドは、その内部で StatCommon メソッドを呼び出しているが、その引数として、変数 data と、$f(x_i)$の実体となる MeanConversion メソッドを渡している。MeanConversion は、Conversion デリゲートと同じシグネチャであるため、StatCommon メソッドの引数 f に代入することができる。MeanConversion メソッドは34-37行目で定義されている。

つまり、Mean メソッドを定義するために、数値を変換する関数を内包するデリゲート型（Conversion 型）を定義し、このデリゲート型の変数を引数とする StatCommon メソッドを定義し、実際に変換を行うため

*5　他の言語でも、関数へのポインタや関数を内包するオブジェクトを引数として渡すことができる。

の関数（MeanConversionメソッド）を定義し、Meanメソッドの中で、StatCommonメソッドを呼び出し、その引数として標本とMeanConversionメソッドを渡して計算を行う、という手続きがとられていることになる。

結果として、従来型のプログラミング言語で上記の設計を実現するためには、非常に冗長な記述が必要となるのである。

第二に、メンバ変数Dataの存在が挙げられる。メンバ変数Dataは、現在計算の対象となっている標本を一時的に保管するものであり、特にVarianceメソッドで計算を行う際に必要とされる。Varianceメソッドでメンバ変数Dataが必要な理由は、プログラミング言語の持つ「スコープ」と呼ばれる仕組みのためである。

プログラミング言語では、ある部分からアクセスできるデータの範囲には制限が課せられている。一般に、関数の内部から直接外部の変数にアクセスすることはできない。ただし、C#のようなオブジェクト指向言語の場合、クラス内のメソッドから同じクラスのメンバ変数にアクセスすることは可能である*6。このような制限をスコープと呼ぶ。つまり、現在考えている仕様でメソッドの外からメソッドの内部にデータを与えるためには、引数かメンバ変数を使う他ない。

ソースコード3では、標本内の数値を変換する関数（VarianceConversionメソッド）が定義されている。この関数は、Varianceメソッド内で、StatCommonメソッドの引数として渡されるものであり、標本内の数値x_iを引数とし$(x_i-\bar{x})^2$という変換を行う。

Varianceメソッド内で計算を始める時点でメンバ変数Dataに標本の値を保管しておくことで（19-20行目）、VarianceConversionメソッドの内部からメンバ変数Dataへアクセスし、$\bar{x}$を計算することが可能となる。

なお、引数を使って、VarianceConversionメソッドの内部に$\bar{x}$を与え

*6　C#2.0以降では、匿名メソッドやラムダ式の内部から、それらが定義されている関数内のローカル変数にアクセスすることが可能である。

ることもできるが、非効率な記述となる可能性が非常に高い。例えば、VarianceConversion メソッドの引数として、標本内の一つの数値だけでなく標本平均も渡すようにした場合、Conversion 型も標本内の一つの数値と標本平均の二つの引数をとるように修正しなくてはならない。結果として、MeanConversion メソッドも同じ二つの引数をとるように再定義することになるが、MeanConversion メソッドは標本平均を必要としないため、不要な引数の受け渡しが発生することになる。

ソースコード４は、F# を使ってソースコード３と同等の処理を行うように記述した例である。

ソースコード４　平均および分散の計算（F#）

```
 1 let data = [1.0; 2.0; 3.0; 4.0; 100.0]
 2
 3 let StatCommon sample f =
 4     List.sumBy f sample / (List.length sample | > float)
 5
 6 let Mean sample =
 7     StatCommon sample (fun x -> x)
 8
 9 let Variance sample =
10     let n = List.length sample | > float
11     StatCommon sample (fun x -> (x - Mean sample) **2.0)
           | > (*) (n / (n - 1.0))
12
13 let xbar = Mean data // val xbar : float = 22.0
14 let s2 = Variance data // val s2 : float = 1902.5
```

第一に F# は関数型プログラミング言語であり、先に述べたように、関数を引数として渡すことが可能である。例えば、ソースコード４の Variance

関数は、StatCommon 関数に、引数として関数$f(x)=(x_i-\bar{x})^2$を渡している。正確には、

```
(fun x -> (x - Mean sample) **2.0)
```

という無名の関数を定義し、それを引数として渡しているのである。つまり、F# では容易に関数を定義することができる。F# において、このような無名の関数を「ラムダ式」と呼ぶ。また、関数を別の変数に割り当てる際、ソースコード３のようにデリゲート型を作成する必要がない。F# では関数は値として扱われるため、その型さえ一致すれば関数を変数に割り当てることができる。結果、F# では、関数の定義や割り当てが容易であるため、従来型のプログラミング言語よりも簡潔にプログラミングを行うことが可能である。

第二に、F# には、関数内部で定義された関数に対して、引数を使わずに外側からデータを渡す仕組みがある。これをクロージャと呼ぶ[*7]。上述したように、ソースコード４の Variance 関数は、StatCommon 関数の引数として（`fun x -> (x - Mean sample) **2.0`）というラムダ式を与えている。すなわち、Variance 関数の内部でさらに無名の関数が定義されていることになる。このラムダ式は、その内部で sample 変数を使って$\bar{x}$の計算を試みているが、sample 変数はラムダ式の外部に存在する変数であるため、従来型の言語ではこのような記述はエラーとなる。しかし、クロージャによりこの処理は F# ではエラーにはならず、正しく計算が行われる。したがって、ソースコード３のように、標本を保管しておくためのメンバ変数を用意する必要がない。

上記の例では、F# を用いることで、より効率的なプログラムの記述が可能となった[*8]。当然ながら、F#、あるいは関数型プログラミング言語を使えば、どんな状況でも効率的な記述ができるわけではない。しかし、関数型

---

*7　F# におけるクロージャについては、Smith（2010）を参照。

*8　C#2.0は関数型プログラミングの機能を一部取り込んでいるため、上記の例の場合、C#2.0より導入された「匿名メソッド」を使うことで、F# の例と同様の記述が可能である。

プログラミングにより、従来の言語では難しかった実装を容易に行える可能性があるといえるだろう。

4　F# におけるオブジェクト指向プログラミング

最後の例として、BFGS 法による最適化を行うプログラムを示す（ソースコード５）[*9][*10]。

ソースコード５　BFGS 法

```
type BFGS (f: (Vector <float > -> float) , iteration: int
    ,tolerance: float) =
    let defaultIteration = 100
    let defaultTolerance = 1e-3
    let lineSearch = LineSearch (f)

    let mutable iteration = iteration
    let mutable tolerance = tolerance
    let mutable derivative = (fun x -> Differentiation.
        slowGradient (f, x))

    let BFGSWeightMatrix (w: Matrix <float >) (xds:
        Vector <float >) (dds: Vector <float >) =
        w + ((1.0 / (dds * xds)) * Vector.OuterProduct(dds
            ,dds)) - ((1.0 / (xds * w * xds)) * Vector.
            OuterProduct ((w * xds) , (w * xds)))
```

*9　このプログラムでは、外部ライブラリである MathNet.Numerics を使用している（http://mathnetnumerics.codeplex.com/）。Vector クラスや Matrix クラスは、MathNet.Numerics で提供されている。また、このプログラムの動作には、別途、線形探索を行うクラス LineSearch や、数値微分を行うクラス Differentiation が必要である。

*10　BFGS 法のアルゴリズムは、Greene（2008）による。

```
12
13      member x.Iteration with get () = iteration and set
            v =iteration <- if v <= 0 then defaultIteration
            else v
14      member x.Tolerance with get () = tolerance and set v
            =tolerance <- if v <= 0.0 then defaultTolerance
            else v
15      member x.Derivative with get () = derivative and set
            v =derivative <- v
16
17      member x.Minimize (initval: Vector <float >) =
18          let w = DenseMatrix.Identity (initval.Count)
19          let ret = initval.Clone ()
20          let grad = derivative ret
21
22          let rec search (w: Matrix <float >) (r: Vector
                <float >) (g: Vector <float >) count =
23              let wi = w.Inverse ()
24
25              if g.Norm (2.0) < x.Tolerance && count > 0
                    then (r, (f r) , wi , true)
26              else if (count >= x.Iteration) then (r, (f r)
                    ,wi , false)
27              else let step = lineSearch.Search ret ((
                    -1.0) * (wi * g))
28                  let newr = r - step * (wi * g)
29                  let newg = derivative newr
30                  let neww = BFGSWeightMatrix w (newr - r)
```

```
                (newg - g)
31                search neww newr newg (count + 1)
32
33          search w ret grad 0
```

先述したように、F# はオブジェクト指向言語でもあるためクラスを定義することが可能である。例えば、ソースコード５では、BFGS という名前のクラスを定義している。そのメンバは、繰り返し回数を保持する変数 Iteration，収束条件を保持する変数 Tolerence，最適化における数値微分の方法を保持する変数 Derivative，および実際に最適化を行う関数 Minimize からなる。

最適化問題に限らず、統計計算では、一つの計算から複数の計算結果が得られることが多い。例えば、BFGS 法による最適化問題の計算結果は、最適解における関数の値、説明変数の値、およびヘッセ行列の近似値となるであろう。これらは、プログラムの内部では、浮動小数点型、ベクトル型、行列型と、異なる型として取り扱われることが一般的であろう。このように、関数の戻り値として型の異なる複数の値を返す必要がある場合、従来のオブジェクト指向言語では、これらの型を内包するクラスを定義しそのインスタンスを戻り値とする方法がとられる。しかし、この方法では戻り値を返すためだけのクラスを定義することになり、冗長な記述が要求される。一方、F# では型の異なる複数の値を一つの組（タプル）にまとめることができる。ソースコード５の Minimize 関数は、最適解の探索が終了したときに、その時点の関数の値、説明変数の値、ヘッセ行列、および収束の正否を一つの組にまとめて返す（25，26行目）。したがって、従来のオブジェクト指向言語と異なり、戻り値のためのクラスを定義する必要がない。

５　まとめ

本稿では、統計計算を例に、関数型プログラミングの機能について説明した。関数型プログラミングにより、従来のプログラミング言語に比べて、処

理に必要な記述が少なくなる。その結果、より効率的かつ迅速なプログラミングが可能となるだけでなく、プログラムの可読性や保守性も向上しうる。特に、F# はマルチパラダイム言語であるため、従来のプログラミング言語と関数型プログラミングのそれぞれが持つ利点を備えているといえる。実際には、ほとんどの言語がマルチパラダイム言語であり、C# のように関数型プログラミングをサポートする言語も増えている。したがって、効率的にプログラムを記述する上で、関数型プログラミングについての理解は極めて重要な要素になりつつあるといえよう。

関数型プログラミング、および F# には、本稿で取り上げた以外にも数多くの機能が存在する。それらの機能を活用することで、さらに効率的なプログラミングが可能となるだろう。

参考文献

Greene, W. H., *Econometric Analysis,* Prentice-Hall, 6th edition, 2008.

Syme, D., A. Granicz, and A Cisternino, *Expert F# 2.0,* Apress, 2010.

C. Smith『プログラミング F#』、オライリージャパン、2010,（鈴木幸敏訳、頃末和義監修）

川俣晶『[完全版] 究極の C# プログラミング－新スタイルによる実践的コーディング』、技術評論社、2009

高橋正子『計算論－計算可能性とラムダ計算』、近代科学社、1991

G. Hutton『プログラミング Haskell』、オーム社、2009,（山本和彦訳）